Rethinking large-scale economic modeling for efficiency: optimizations for GPU and Xeon Phi clusters

Kübler, Felix ; Mikushin, Dmitry ; Scheidegger, Simon ; Schenk, Olaf

Abstract: We propose a massively parallelized and optimized framework to solve high-dimensional dynamic stochastic economic models on modern GPU- and MIC-based clusters. First, we introduce a novel approach for adaptive sparse grid index compression alongside a surplus matrix reordering, which significantly reduces the global memory throughput of the compute kernels and maps randomly accessed data onto cache or fast shared memory. Second, we fully vectorize the compute kernels for AVX, AVX2 and AVX512 CPUs, respectively. Third, we develop a hybrid cluster oriented work-preempting scheduler based on TBB, which evenly distributes the time iteration workload onto available CPU cores and accelerators. Numerical experiments on Cray XC40 KNL "Grand Tave" and on Cray XC50 "Piz Daint" systems at the Swiss National Supercomputer Centre (CSCS) show that our framework scales nicely to at least 4,096 compute nodes, resulting in an overall speedup of more than four orders of magnitude compared to a single, optimized CPU thread. As an economic application, we compute global solutions to an annually calibrated stochastic public finance model with sixteen discrete, stochastic states with unprecedented performance. Index Terms—High-Performance Computing, Macroeconomics, Public Finance, Adaptive Sparse Grids, Heterogeneous Systems, CUDA, GPU, MIC

DOI: https://doi.org/10.1109/IPDPS.2018.00070

Rethinking large-scale economic modeling for efficiency: optimizations for GPU and Xeon Phi clusters

Simon Scheidegger*†, Dmitry Mikushin*‡, Felix Kübler*§. Olaf Schenk¶‖
*Department of Banking and Finance, University of Zurich, Switzerland
‖Institute of Computational Science, Faculty of Informatics, Universit a della Svizzera italiana, Switzerland
†simon.scheidegger@bf.uzh.ch
‡dmitry.mikushin@bf.uzh.ch
§felix.kuebler@bf.uzh.ch
¶olaf.schenk@usi.ch

***Abstract*—We propose a massively parallelized and optimized framework to solve high-dimensional dynamic stochastic economic models on modern GPU- and MIC-based clusters. First, we introduce a novel approach for adaptive sparse grid index compression alongside a surplus matrix reordering, which significantly reduces the global memory throughput of the compute kernels and maps randomly accessed data onto cache or fast shared memory. Second, we fully vectorize the compute kernels for AVX, AVX2 and AVX512 CPUs, respectively. Third, we develop a hybrid cluster oriented work-preempting scheduler based on TBB, which evenly distributes the time iteration workload onto available CPU cores and accelerators. Numerical experiments on Cray XC40 KNL "Grand Tave" and on Cray XC50 "Piz Daint" systems at the Swiss National Supercomputer Centre (CSCS) show that our framework scales nicely to at least 4,096 compute nodes, resulting in an overall speedup of more than four orders of magnitude compared to a single, optimized CPU thread. As an economic application, we compute global solutions to an annually calibrated stochastic public finance model with sixteen discrete, stochastic states with unprecedented performance.**

***Index Terms*—High-Performance Computing, Macroeconomics, Public Finance, Adaptive Sparse Grids, Heterogeneous Systems, CUDA, GPU, MIC**

I. Introduction

Optimal taxation and the optimal design of public pension systems are classic themes in economics with obvious relevance for society. To address these questions quantitatively, dynamic stochastic general equilibrium models with heterogeneous agents are used for counter-factual policy analysis. One particular subclass is called an overlapping generation (OLG) model [1]. These models are essential tools in public finance since they allow for careful modeling of individuals' decisions over the life cycle and their interactions with capital accumulation and economic growth.

There are now several areas where, over the last 10 to 20 years, deterministic OLG models have been fruitfully applied to the analysis of taxation and fiscal policy. In particular large-scale deterministic versions of the model have been applied to the "fiscal gap" [2], to "dynamic scoring of tax policies" [3], and to the evaluations of social security reforms (see, e.g., [4]). It is clear, however, that to be able to address these policy-relevant questions thoroughly, uncertainty needs to be included in the basic model. Both uncertainty about economic fundamentals as in [5] as well as uncertainty about future policy [6] crucially affect individuals' savings, consumption, and labor-supply decisions and the uncertainty in the specification of the model can overturn many results obtained in the deterministic model. Moreover, uncertainty about future productivity as well as uncertainty about future taxes have first-order effects on agents' behavior. Unfortunately, when one introduces this form of uncertainty into the model, there does not exist steady-state equilibria, as the stochastic aggregate shocks affect everybody's return to physical and human capital. These effects do not cancel out in the aggregate so that the distribution of wealth across generations changes with the stochastic aggregate shock. This feature makes it difficult to approximate equilibria with many agents of different ages and aggregate uncertainty—realistic calibrations of the model lead to very-high-dimensional problems that were so far thought to be unsolvable. This explains why relatively little policy-work has been carried out using stochastic OLG models. Krueger and Kubler [7], for example, analyze welfare implications of social security reforms in an OLG model where one period corresponds to six years, thereby reducing the number of adult cohorts and thus the dimensionality of the problem by a factor of six. Hasanhodzic and Kotlikoff [8], on the other hand, approximate the solution of an OLG model using simulation-based methods and certainty equivalents. Their method only yields acceptable solutions for special cases and cannot be easily extended to tackle general OLG models.

This article shows how we can leverage recent developments in computational mathematics and massively parallel hardware to compute global solutions to general stochastic OLG models in relatively short times. As a test case, we have solved a 59-dimensional model with 16 discrete, stochastic states—much larger than any problem known to be solved so far in this stream of the literature. Therefore, our methodology opens the room to address economic research questions of unprecedented realism.

In stochastic dynamic models, individuals' optimal policies and prices are unknown functions of the underlying, high dimension states and are solved for by so-called *time iteration* algorithms (see, e.g., [9]). Two major bottlenecks create difficulties in achieving a fast time-to-solution process when

solving large-scale dynamic stochastic OLG models with this iterative method, namely,

(i) in each iteration step, several economic functions need to be approximated and interpolated. For this purpose, the function values have to be determined at many points in the high-dimensional state space, and
(ii) each point involves solving a system of nonlinear equations (around 60 equations in 60 unknowns).

We overcome these difficulties by massively reducing the number of grid points required to represent the economic functions by using adaptive sparse grids (ASGs; see, e.g., [10]–[13]) as well as by compressing the ASGs only to visit points with meaningful contribution when interpolating on them. Also, the time spent in each iteration step is substantially reduced by applying massively parallel processing. Using the Message Passing Interface (MPI) [14], we distribute the workload—that is, the grid points, across compute nodes. The nodes can optionally be equipped with NVIDIA GPUs. Within a single node, the workload is further partitioned among CPU cores and GPU with Intel Thread Building Blocks (TBB) [15]. The code for CPU deploys AVX, AVX2 or AVX-512 vectorization for Sandy/Ivy Bridge, Haswell/Broadwell or Skylake/KNL, respectively, while NVIDIA GPU kernels are written in CUDA [16]. This scheme enables us to make efficient use of the contemporary HPC facilities that consist of a variety of special purpose as well as general purpose hardware and whose performance nowadays can reach dozens of petaflop/s (https://www.top500.org). To sum up, the main contributions of this paper are as follows:

- Building on [17], [18], we propose a generic parallelization scheme for time iteration algorithms that aim to solve mixed high-dimensional continuous/discrete state dynamic stochastic economic models.
- We show that our parallelization approach is ideally suited for heterogeneous CPU/GPU HPC systems as well as for Intel Xeon Phi KNL clusters.
- We introduce an original compression method for ASGs that reduces computations, yet allowing partial vectorization and randomly accessed data fitting into cache or GPU shared memory.
- We present highly efficient and scalable implementations of the time iteration algorithm on the aforementioned hardware platforms.
- As an example application, we compute global solutions to 59-dimensional OLG model with 16 discrete, stochastic states with unprecedented performance.

This paper is organized as follows. In Sec. II, we describe the abstract economic models we aim to solve. In Sec. III, we briefly summarize the theory of ASGs. In Sec. IV, we embed ASG interpolation in a time iteration algorithm. Moreover, we also discuss the respective hybrid parallelization scheme as well as a novel compression method for ASGs that substantially reduces computations. In Sec. V, we report on how our implementation performs and scales in solving an annually calibrated, stochastic OLG model.

II. Overlapping generation models

To demonstrate the capabilities of our method, we consider an annually calibrated, stochastic OLG model similarly to the one described in [5]—that is, agents have a model lifetime of 60 periods, each corresponding to one year of life after the age of 20. Moreover, there are $N_s = 16$ discrete states in our model that represent the economy in a variety of situations such as booms, busts as well as different tax regimes. Agents face taxes τ_l on labor income and τ_c on capital income. Tax rates change stochastically over time and are used to fund a pay-as-you-go social security system. We assume that the average retirement age is 65, and that agents receive social security payments, financed by the labor-income tax, starting at age 66. It is clear that the level of complexity listed here is needed to model for example demographic effects that are caused by retirement as well as to mimic the fact that agents choose their actions based on expectations about an uncertain future. However, taken together, this all results in a very intricate formal structure of the model.

This is an example of a broad class of models in macroeconomics and public finance and is typically solved by time iteration algorithms (see, e.g., [9]). To this end, we outline in Sec. II-A the general structure that is common to OLG models. Moreover, we briefly describe how we iteratively solve them.

A. Abstract model formulation and solution method

The formal structure[1] common to stochastic OLG models can be described as follows: The economy is populated with agents that live for A periods. Each of them can uniquely be identified by her age a, where $1 \leq a \leq A$. Let $s_t = (z_t, x_t) \in S \subset \mathbf{Z} \times \mathbf{B} \subset \mathbb{R} \times \mathbb{R}^{\mathbf{d}}$ denote the state of the economy at time $t \in \mathbb{N}$, where $\mathbf{Z}$ is a finite set of size $N_s \in \mathbb{N}$, $d = A-1$ is the dimensionality of x_t, and $\mathbf{B}$ is a $d-$dimensional rectanglular box. z_t represents a stochastic shock to the economy, e.g., to it's output, and x_t characterizes the economy in z_t. In our OLG model, it is given by

$$x_t = (K, \omega_2, ..., \omega_{A-1}) \subset \mathbb{R}^{A-1}, \tag{1}$$

where K is the aggregate capital and ω_i are the wealth levels of generations $i = 2$ to $i = A-1$. The actions of all agents in the economy can be represented by a *policy function* $p : S \to Y$, where Y is the space of possible *policies*. In our OLG model, the optimal policy $p : R^{N_s \cdot d} \to R^{N_s \cdot 2 \cdot d}$ maps the current state s_t into unknown asset demand functions $k_i : \mathbf{Z} \times \mathbf{B} \to \mathbb{R}$ and value functions $v_i : \mathbf{Z} \times \mathbf{B} \to \mathbb{R}$, where $i = 1, ..., A-1$, and $z \in \mathbf{Z}$. Furthermore, the evolution of the current state of the economy s_t from period t to $t+1$ is described by the state transition

$$s_{t+1} \sim \mathcal{P}\left(\cdot | s_t, p(s_t)\right), \tag{2}$$

where the distribution $\mathcal{P}(\cdot)$ is pre-defined and model specific. In our case, the stochastic transition of the economy from period t to $t+1$ is given by a Markov chain—that

[1] Note that we omit a detailed discussion of the OLG model, as this is beyond the scope of the paper. For a detailed review of this application, we refer to [5].

is, z_t follows a first-order Markov process with transition probability $\pi(z'|z)$. The stationary policy function p needs to be determined from *equilibrium conditions*. These conditions constitute a functional equation that the policy function p has to satisfy, namely, that for all $s_t \in S$,

$$0 = \mathbb{E}\left[f\Big(s_t, s_{t+1}, p\left(s_t\right), p\left(s_{t+1}\right)\Big)|s_t, p(s_t)\right], \tag{3}$$

where $f : S^2 \times Y^2 \to \mathbb{R}^{N_s \cdot d}$ represents the period-to-period equilibrium conditions of the OLG model, and where the expectation operator is taken on the the disrete shocks. This function is nonlinear because of concavity assumptions on utility and production functions. As a direct consequence, the optimal policy p solving (3) will also be nonlinear. Hence, approximating it only locally might provide misleading results. For such applications, we, therefore, need a global solution, that is, we need to approximate p over the entire state space S. In our work, we approximate the unknown equilibrium asset demand and value functions on an individual ASG per discrete state $z \in \mathbf{Z}$ by piecewise multilinear functions $\hat{k}_i(z, \cdot|\alpha^k)$, $\hat{v}_i(z, \cdot|\alpha^v)$ that are uniquely defined by finitely many coefficients α^k, α^v (see Sec. III). In order to solve for the unknown coefficients, we require that the functional equations of the OLG model (see (3) and [5]) hold exactly at M grid points $x_{i=1,...,M} \in \mathbf{B}$ per discrete state z.

Our computational strategy to solve the OLG model is to search for a recursive equilibrium (see, e.g., [19])—that is, a time-invariant policy function p by using a time iteration algorithm (see, e.g., [9]). The sequential version of this algorithm is summarized in code listing 1 and is based on the following heuristic: solve the equilibrium conditions of the model for today's policy $p : S \to Y$ taking as given an initial guess for the function that represents next period's policy, p_{next}; then, use p to update the guess for p_{next} and iterate the procedure until numerical convergence is reached. As a practical consequence, we need to compute many successive approximations of p that rely on interpolating on p_{next}. To do this efficiently, we employ ASGs (see section III) in combination with a hybrid parallelization scheme (see section IV-A) as well as a novel ASG compression scheme (see section IV-B).

Data: Initial guess for $p = (p(z = 1), ..., p(z = N_s))$.
Convergence tolerance tol.
Result: The time-invariant policy function p.
while $\epsilon > tol$ **do**
  $p_{next} \leftarrow p$.
  for $z = 1; z \leq N_s;\ z = z + 1$ **do**
    approximate $p(z)$ by solving (3) at M grid points **given** p_{next}.
  end
  $\epsilon = \|p - p_{next}\|$.
end

Algorithm 1: Time iteration algorithm.

III. Basics on adaptive sparse grids

Our method of choice to tackle the numerical issues that arise from the nature of the high-dimensional state space as described in Sec. II, namely, the repeated construction and evaluation of multivariate policy and value functions (see (3)) are ASGs. In this section, we summarize its basics. For thorough derivations, we point the reader, e.g., to [10], [20].

We consider the representation of a piecewise d-linear function $f : \Omega \to \mathbb{R}$ for a certain mesh width $h_n = 2^{1-n}$ with some discretization level $n \in \mathbb{N}$. As we aim to discretize Ω, we restrict our domain of interest to the compact sub-volume $\Omega = [0, 1]^d$, where d in our case is the dimensionality of the OLG model. This situation can be achieved for most other domains by re-scaling and possibly carefully truncating the original domain. In order to generate an approximation u of f, we construct an expansion

$$f\left(\vec{x}\right) \approx u\left(\vec{x}\right) := \sum_{j=1}^{N} \alpha_j \phi_j\left(\vec{x}\right) \tag{4}$$

with N basis functions ϕ_j and coefficients α_j. We use one-dimensional hat functions

$$\phi_{l,i}(x) = \begin{cases} 1, & l = i = 1, \\ \max(1 - 2^{l-1} \cdot |x - x_{l,i}|, 0), & i = 0, ..., 2^{l-1},\ l > 1, \end{cases} \tag{5}$$

which depend on a level $l \in \mathbb{N}$ and index $i \in \mathbb{N}$. The corresponding grid points are distributed as

$$x_{l,i} = \begin{cases} 0.5, & l = i = 1, \\ i \cdot 2^{1-l}, & i = 0, ..., 2^{l-1},\ l > 1, \end{cases} \tag{6}$$

and are depicted in Fig. 1. We use a sparse grid interpolation method that is based on a hierarchical decomposition of the underlying approximation space. Hence, we next introduce, hierarchical index sets I_l:

$$I_l := \begin{cases} \{i = 1\}, & \text{if } l = 1, \\ \{0 \leq i \leq 2,\ i \text{ even}\} & \text{if } l = 2, \\ \{0 \leq i \leq 2^{l_t - 1},\ i \text{ odd}\} & \text{else}, \end{cases} \tag{7}$$

that lead to hierarchical subspaces W_l spanned by the corresponding basis $\phi_l := \{\phi_{l,i}(x), i \in I_l\}$. See Fig. 1 for the basis functions up to level 3. The hierarchical basis functions extend to the multivariate case by using tensor products:

$$\phi_{\vec{l},\vec{i}}\left(\vec{x}\right) := \prod_{t=1}^{d} \phi_{l_t,i_t}\left(x_t\right), \tag{8}$$

where $\vec{l}$ and $\vec{i}$ are multi-indices, uniquely indicating level and index of the underlying one-dimensional hat functions for each dimension. They span the multivariate subspaces by

$$W_{\vec{l}} := \text{span}\{\phi_{\vec{l},\vec{i}} : \vec{i} \in I_{\vec{l}}\} \tag{9}$$

with the index set $I_{\vec{l}}$ given by a multidimensional extension to (7):

$$I_{\vec{l}} := \begin{cases} \{\vec{i} : i_t = 1, 1 \leq t \leq d\} & \text{if } l = 1, \\ \{\vec{i} : 0 \leq i_t \leq 2,\ i_t \text{ even}, 1 \leq t \leq d\} & \text{if } l = 2, \\ \{\vec{i} : 0 \leq i_t \leq 2^{l_t - 1},\ i_t \text{ odd}, 1 \leq t \leq d\} & \text{else.} \end{cases} \tag{10}$$

The space of piecewise linear functions V_n on a Cartesian grid with mesh size h_n for a given level n is then defined by the direct sum of the increment spaces (cf. (9)):

$$V_n := \bigoplus_{|l|_\infty \leq n} W_{\vec{l}}, \quad |l|_\infty := \max_{1 \leq t \leq d} l_t. \tag{11}$$

The interpolant of f, namely, $u(\vec{x}) \in V_n$, can now uniquely be represented by

$$f(\vec{x}) \approx u(\vec{x}) = \sum_{|l|_\infty \leq n} \sum_{\vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \cdot \phi_{\vec{l},\vec{i}}(\vec{x}). \tag{12}$$

Note that the coefficients $\alpha_{\vec{l},\vec{i}} \in \mathbb{R}$ are commonly termed hierarchical surpluses. They are merely the difference between the function values at the current and the previous interpolation levels. For a sufficiently smooth function f the asymptotic error decays as $\mathcal{O}\left(h_n^2\right)$ but at the cost of spending $\mathcal{O}\left(h_n^{-d}\right) = \mathcal{O}\left(2^{nd}\right)$ grid points, thus suffering the curse of dimensionality [21]. As a consequence, the question that needs to be answered is how we can construct discrete approximation spaces that are better than V_n in the sense that the same number of invested grid points leads to a higher order of accuracy. Luckily, for functions with bounded second mixed derivatives, it can be shown that the hierarchical coefficients rapidly decay, namely, $|\alpha_{\vec{l},\vec{i}}| = \mathcal{O}\left(2^{-2|\vec{l}|_1}\right)$. Hence, the hierarchical subspace splitting allows us to select those $W_{\vec{l}}$ that contribute most to the overall approximation. This can be done by an a priori selection, resulting in the sparse grid space V_n^S of level n, defined by

$$V_n^S := \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}}, \quad |\vec{l}|_1 = \sum_{i=1}^{d} l_t. \tag{13}$$

In Fig. 1, we depict its construction for $n = 3$ in two dimensions. V_3^S shown there consists of the hierarchical increment spaces $W_{(l_1,l_2)}$ for $1 \leq l_1, l_2 \leq n = 3$. The number of grid points required by the space V_n^S is now of order $\mathcal{O}\left(2^n \cdot n^{d-1}\right)$, which is a significant reduction of the number of grid points, and thus of the computational and storage requirements compared to the Cartesian grid space. In analogy to (12), a function $f \in V_n^S \subset V_n$ can now be expanded by

$$f(\vec{x}) \approx u(\vec{x}) = \sum_{|l|_1 \leq n+d-1} \sum_{\vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \cdot \phi_{\vec{l},\vec{i}}(\vec{x}), \tag{14}$$

which contains substantially fewer terms. In the case that functions do not meet the smoothness requirements or that show distinct local features as we face in the model described in Sec. II, they can still be tackled efficiently with sparse grids if spatial adaptivity is used. The classical sparse grid construction introduced in (13) defines an a priori selection of grid points that are optimal for functions with bounded second-order mixed derivatives. An adaptive (a posteriori) refinement can, additionally, based on a local error estimator, select which grid points in the sparse grid structure should be refined. The most common way of doing so is to add $2d$ children in the hierarchical structure with increasing grid refinement level if the hierarchical surpluses satisfy $g(\alpha) \geq \epsilon$ for a so-called refinement threshold $\epsilon \geq 0$. For more details regarding ASGs, we refer the reader e.g. to [11], [12], [22]. The lower right panel of Fig. 1 illustrates a qualitative example of how a sparse grid is refined adaptively, adding a second layer of sparsity to the sparse grid.

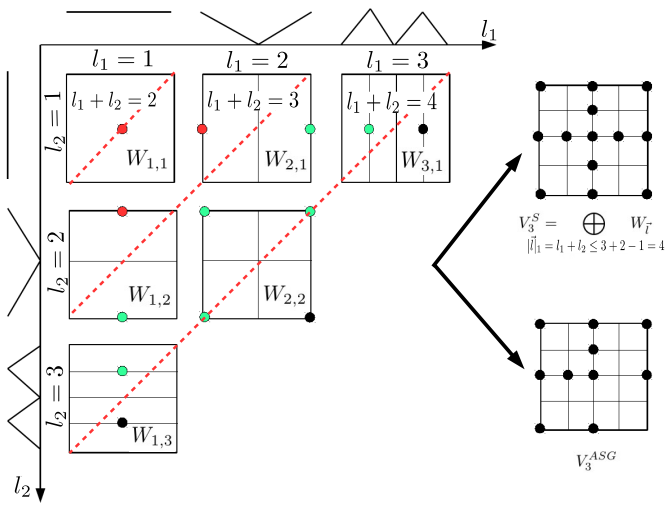


Fig. 1: Left panel: Hierarchical increment spaces $W_{(l_1,l_2)}$ for $1 \leq l_1, l_2 \leq n = 3$ with their corresponding grid points and one-dimensional piecewise linear basis functions of levels 1, 2, and 3. Top right panel: Construction of a classical sparse grid V_3^S (see (13)), consisting of all the points displayed in the left figure. Note that the optimal selection of subspaces for the classical sparse grid is indicated by the dashed lines of constant $l_1 + l_2$. Bottom right panel: Construction of the ASG V_3^{ASG}. Note that the red dots in the left panel symbolically represent points that would be refined, i.e., $g(\alpha) \geq \epsilon$ holds, whereas the green ones indicate points where the grid is not further refined. The black points in the left panel are only contained in V_3^S, and not V_3^{ASG}.

IV. Parallel time iteration algorithm

We describe now how to solve the stochastic OLG model introduced in Sec. II. For this reason, we implement a massively parallel version of a time iteration algorithm (see code listing 1 and [9]) for mixed high-dimensional continuous/discrete states that uses one ASG per shock $z \in \mathbf{Z}$ in each iteration step.[2] Building on [18], we parallelize this algorithm by a hybrid scheme using MPI [14], TBB [15] (and CUDA, if a GPU is present on the compute node), and deploy compute kernels that leverage AVX, AVX2 or AVX-512 vectorization, depending on

[2]In line with Sec. II, the mixed discrete/continuous state variables of the OLG model at time t consist of $s = (z, x)$, where x has 59 dimensions, and the shock z has $N_s = 16$ possible realizations. Moreover, the policy function $p = (p(z = 1, \cdot), \ldots, p(z = 16, \cdot)) : \mathbb{R}^{16 \cdot 59} \to \mathbb{R}^{16 \cdot 2 \cdot 59}$ maps the current state into asset demand and value functions (see Sec. II). We, therefore, have to approximate 118 coefficients $\alpha = (\alpha^k, \alpha^v)$ per state z and grid point.

the respective hardware platform (see Sec. IV-A).[3]

One significant performance bottleneck when solving large-scale economic models always lie on interpolating the previous iteration step's policy functions. When searching for the solution to the equation system at a given point for a given shock z (cf. (3)), the algorithm has to frequently interpolate on the policy functions p_{next} of all the $N_s = 16$ states from the previous iteration step at once. These interpolations typically take up to 99% of the computation time needed (see, e.g., [17]) to solve the nonlinear set of equations and therefore need to be carried out as rapidly as possible to guarantee a fast time-to-solution process. In our earlier work [18], we applied a dense matrix data format that is very similar to the one proposed by [23] and for which highly optimized algorithms exist to perform the interpolation task. However, for the applications in scope here, we cannot maintain this data structure, since in contrast to [18], where we had to deal with interpolating on one single ASG of intermediate size only (around 8 continuous dimensions), we now have to be able to operate on 16 very large—that is, 59-dimensional ASGs at once (cf. Secs. II and V). Keeping the aforementioned dense matrix format to store the previous timestep's policy function for the interpolation introduces a memory footprint of a non-trivial size that, in turn, would substantially slow down interpolations and thus the time-to-solution. To this end, we propose a novel, generic data compression method for ASGs (see Sec. IV-B).

A. Hybrid parallelization scheme on heterogeneous HPC systems

In every step i of the time iteration procedure (see code listing 1), the policy function p is updated by using a hybrid-parallel algorithm (see Fig. 2). Conceptually, the top layer of parallelism is the N_s discrete states of the OLG model, which are completely independent of each other within a time step. Hence, the `MPI_COMM_WORLD` communicator is split into $N_s = 16$ sub-communicators, each of them representing an individual discrete state—that is, an independent ASG which updates its share of the total policy $p = (p(z = 1), \ldots, p(z = N_s))$. Next, every `MPI_Group` gets assigned a fraction from all the MPI processes available in the `MPI_COMM_WORLD` communicator such that an optimal workload balance across different discrete states in guaranteed[4] (see the top part of Fig. 2). We achieve this by using the number of grid points M_z contained in $p_{next}(z)$ from the previous iteration step $i - 1$ as a proxy for the demand on computational resources necessary in the current time step i. In particular, we assign the fraction $\texttt{MPI\_COMM\_SIZE}(z) = M_z / \left(\sum_{j=1}^{N_s} M_j\right)$ from the total available MPI processes to an individual state z.[5] Inside every `MPI_Group`, an ASG is constructed in a massively parallel fashion. The points that are newly generated within a refinement level (see (13)) are distributed via MPI among multiple, multi-threaded processes. The points that are sent to one particular compute node are then further distributed among different threads. Multithreading on compute nodes is implemented with TBB. To guarantee efficient use of any of the compute nodes, the threads leverage TBB's automatic workload balancing based on stealing tasks from the slower workers. In general, each TBB thread has to solve an independent set of nonlinear equations for every single grid point assigned to it. These nonlinear equations (see (3)) are solved with `Ipopt` [24], which is high-quality open-source software for solving nonlinear programs (`http://www.coin-or.org/Ipopt/`). On top of this, we add an additional level of parallelism. When searching for the solution to the equation system at a given point for a given shock z, the algorithm has to frequently interpolate on the policy functions of all the $N_s = 16$ states from the previous iteration step at once. As they have a high arithmetic intensity—that is to say, many arithmetic operations are performed for each byte of memory transfer and access—they can leverage on SIMD AVX, AVX2 and AVX-512 instructions as well as on the massive parallelism of GPUs, depending on the hardware we deploy our code framework on. In the case of CPU/GPU nodes, we offload parts of the policy function interpolation from the compute nodes to their attached accelerators. In particular, one of the TBB-managed threads is exclusively used for the GPU dispatch, as indicated in the lower part of Fig. 2.

B. Adaptive sparse grid compression

While the primary arithmetic operations to calculate surpluses and perform interpolations on ASGs are rather simple (see Sec. III), accessing the data requires most of the computing time, which emphasizes the importance of efficient data structures. Depending on the target hardware platform, the most widespread techniques for storing ASGs are matrix-kind of structures (see, e.g., [23]) or hash tables (see, e.g., [22], and references therein). However, a direct application of those schemes is suboptimal due to particularities of the target application. To reduce the compute time spent on interpolations when performing time iteration, we, therefore, introduce here a novel data compression scheme for ASGs. Its primary features are that it significantly reduces the global memory throughput of the compute kernels and maps randomly accessed data onto cache or fast shared memory. Conceptually, an ASG is represented by a set of nno points that are all uniquely defined by multi-index pairs $(\vec{l}, \vec{i})$ as well as a vector of surpluses $\vec{\alpha}$ (see Eqs. 10 and 14). Let $\widetilde{\Xi}$ be a matrix that is formed of

[3]Note that the developments presented in this article substantially improve over our previous work [18]. First, we extend our original code base, and it's respective parallelization scheme such that it can handle high-dimensional continuous as well as discrete, stochastic states at the same time. Second, our compute kernels now also deploy AVX2 and AVX-512 vectorization. Third, we introduce a novel data structure for operating on ASGs (see Sec. IV-B).

[4]Note that the overhead of invoking an `MPI_barrier` after each iteration to synchronize across sub-communicators (see Fig. 2) is typically relatively small—that is, less than 1% of the total runtime.

[5]As a simple example, assume that $N_s = 2$, $p_{next}(z = 1)$ consists of 200 points and $p_{next}(z = 2)$ contains 100 points. Moreover, assume that there are 3 MPI processes available in `MPI_COMM_WORLD`. In that case, 2 MPI processes are assigned to `MPI_Group_1`, whereas `MPI_Group_2` receives 1 MPI process.

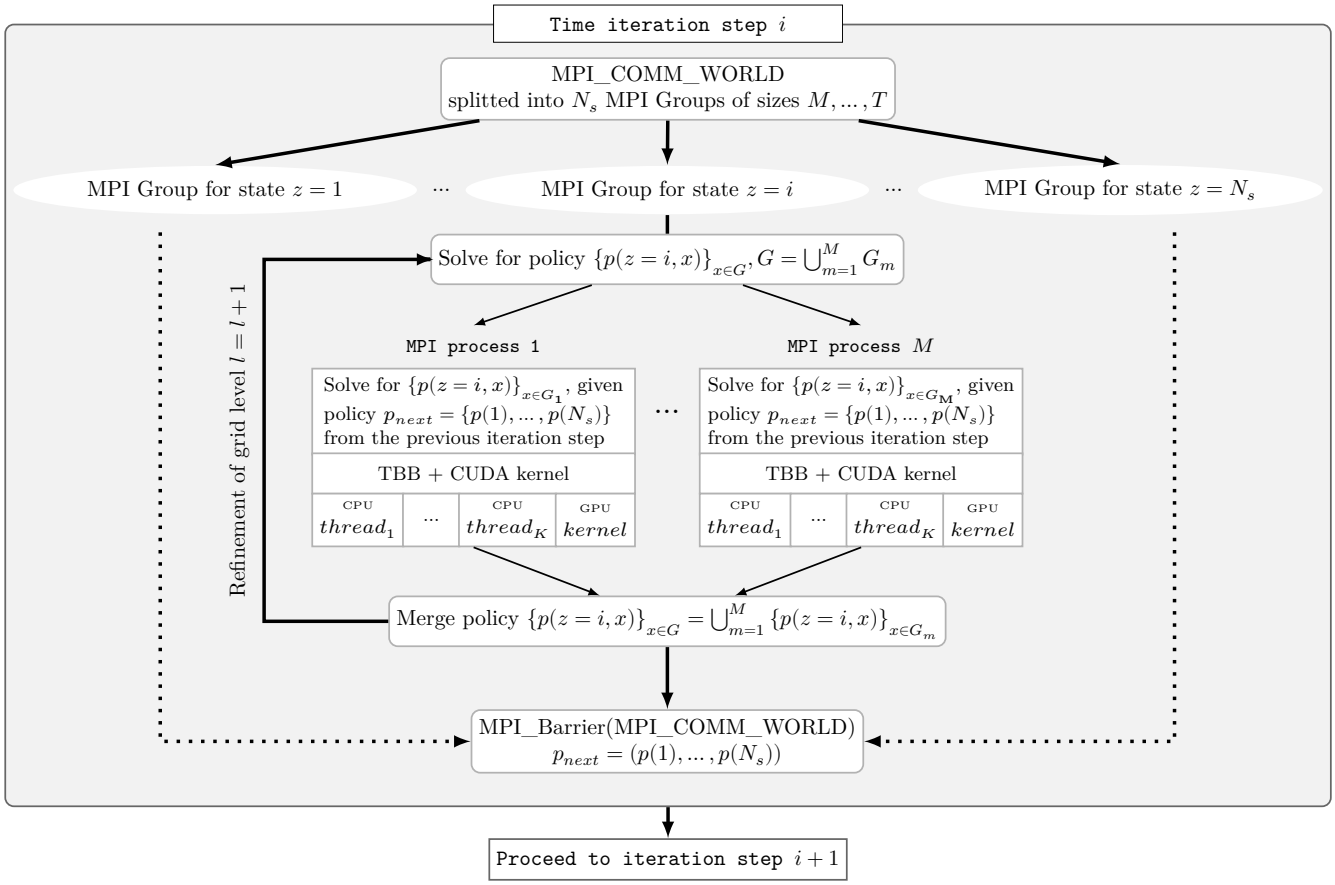
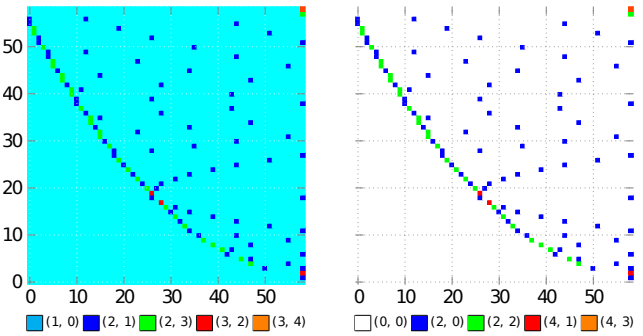


Fig. 2: Schematic representation of the hybrid parallelization scheme in a single time step. Every `MPI` process within an `MPI_Group` is using TBB. In the case of deploying our software on hybrid CPU/GPU nodes, the interpolation on the next period's policy function p_{next} is partially offloaded to GPUs.



(a) Initial (l, i) data. (b) Zero-eliminated (l, i) data.

Fig. 3: First step of the data compression: initial elimination of zeros in $\widetilde{\Xi}$, examplified by a $(0 \dots 58) \times (0 \dots 58)$ submatrix for a sparse grid of maximum refinement level 2. In both figures, the x-direction corresponds to individual l-values, whereas the y-axis corresponds to individual i-values.

multi-index pairs, as illustrated in Fig. 3a by an example for a sparse grid of maximum refinement level 2.[6] Next, we derive a matrix Ξ from $\widetilde{\Xi}$ by pre-processing the scalar entries for every dimension t of the multi-index pairs as $l \leftarrow 2 \ll (l - 2)$ and $i \leftarrow i - 1$, which leads up 96.8% of "zeros" content, as shown in Fig. 3b. Note that the (l, i) pair in a given dimension t is considered zero only if both l and i are zero at the same time. Depending on the level l, each d-sized row usually contains only few non-zero pairs: at most 1 for level 1, and at most 2 for level 2. Moreover, let n_{freq}—that is, *the number of freqencies*, be the maximum number of non-zero values across individual Ξ rows. We decompose the dense Ξ matrix into a set of matrices $\xi_{freq}^{?\times d}$, where $freq = \overline{1, n_{freq}}$.[7] Each of those matrices shall contain no more than one non-zero element from each Ξ row such that the sequences of elements picked up as one from every ξ_{freq}, could, later on, be built up into *chains*. Therefore, ξ_{freq} rows may still

[6] Throughout this section, we count the sparse grid "level" l in *C++* style, starting from $l = 0$ and not $l = 1$, as we do in the remainder of the article.

[7] $\xi_{freq}^{?\times d}$ is a short-hand notation for the fact that we have a dynamically expandable matrix with fixed row size. We start filling it will elements, starting from the first row. If the first row's j-th column is already busy, we append the second row and place another j-th element there, and so on.

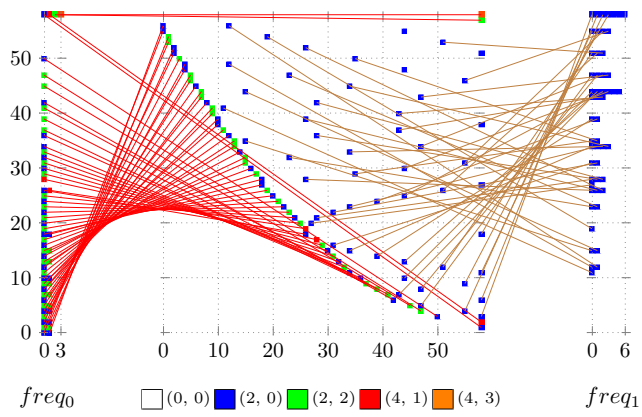


Fig. 4: Second step of the data compression: distribution of the non-zeros from Ξ across two tables of $\{(l,i), k_{nno}\}$ elements. In addition to (l,i), the pair's Ξ row index is stored into k_{nno}

contain some zero elements. The number of rows in $\xi_{freq}^{?\times d}$ is dynamically expanded to fit the actual non-zero population. The set of ξ_{freq} matrices is a sparse representation of Ξ—each ξ_{freq} element, in addition to (l,i), holds the pair's row index in Ξ. Fig. 4 illustrates the decomposition of Ξ into two ξ_{freq} matrices. Note that for illustrative purposes, we show here only the first 59 points from the example sparse grid being represented in our novel data structure. After the initial placement of its elements, the Ξ row index components from each ξ_{freq} matrix element are renumbered in a sorted order that ranges from the first to the last row of ξ_{freq}. A set of *transition* matrices T_{freq} holds correspondences between row indices of consecutive ξ_{freq} matrix pairs after the individual renumbering. The original Ξ row indices in the $\xi_{freq}-$elements are omitted after renumbering. The elements of the ξ_{freq} matrices are further iterated to form a global array of unique elements xps, and a linear lookup index vector V_{freq} is defined for each ξ_{freq} matrix. As result, the size of the xps array denotes which of the linear basis calculations have a non-zero contribution in forming the ASG interpolant (see (14)) and thus are meaningful to perform. Finally, we use T_{freq}, xps, and the lookup indices V_{freq} to construct the set of contributing linear basis chains, as shown in code listing 2). Note that the rows from the matrix in which we store the

for $i = 0,\ i_{ch} = 0;\ i < nno;\ i = i + 1,\ i_{ch} = i_{ch} + n_{freq}$ **do**
  $chains\ (i_{ch})$ = $V(0, i)$;
  for $i_{freq} = 0;\ i_{freq} < n_{freq};\ i_{freq} = i_{freq} + 1$ **do**
    $chains\ (i_{ch})$ = $V(i_{freq}, T(i_{freq}, i))$;
  end
end

Algorithm 2: Construction of chains from transition matrices and lookup indices.

hierarchical surpluses are reordered accordingly.

Our main motivation for the sparse grid index compression introduced above is to eliminate redundant computations when interpolating on the ASG (see (14)). Indeed, as shown in the left panel of Fig. 5 by example of a pure x86 (serial) code listing, the complexity of the linear basis computation shrinks from $nno \times d$ iterations in the dense representation (see [18]) down to $nno \times n_{freq}$ in the case of our proposed data format. Given that in our practital application (see Sec. V), $d = 59$ and n_{freq} is a small constant ($n_{freq} \leq 7$ in typical cases), the complexity goes down by about one order of magnitude, yet introducing some memory access penalty due to additional indexing chains. The number of meaningful basis function factors xps to be calculated is usually relatively small. For instance, $xps = 473$ in the case of a sparse grid that consists of about $300,000$ points (see Fig. I), which easily fits the cache as well as the GPU shared memory (48 KB). In Sec. V-A, we analyze the overall performance impact of the index compression for different interpolation kernels.

test	d	nno	level	# states	# xps/state
"7k"	59	7,081	3	16	237
"300k"	59	281,077	4	16	473

TABLE I: Interpolation test cases for varying sparse grid levels.

V. Performance and Scaling

In this section, we first show in Sec. V-A how the data structure introduced in Sec. IV-B improves on the performance of the interpolation kernels. Second, we report in Sec. V-B on the single node performance achieved by the entire time iteration algorithm. Third, we evaluate the strong scaling behavior of our implementation in section V-C. Finally, we discuss solutions to a public finance OLG model in section V-D.

We deploy our code on two different types of hardware. As the first testbed, we use the Cray XC50 "Piz Daint" system. Cray XC50 compute nodes combine Intel Xeon E5-2690 v3 CPUs with one NVIDIA P100 GPU.[8] Second, we use the Cray XC40 Iron Compute "Grand Tave" system, whose nodes consist of Intel Xeon Phi 7230.[9]

A. Performance of the interpolation kernels

To demonstrate the performance gains of the interpolation kernels with respect to the novel data format (see Sec. IV-B), we carried out two tests on ASGs of varying size. The detailed specification for each of the test cases are summarized in Tab. I). Below, we first give a short description of every version of the interpolation kernel (cf. (14)) and then subsequently report on the achieved performance.

gold: The *gold* version denotes a scalar interpolation kernel that operates on the data format we were using in [18] and which was based on [23].

x86: The *x86* version leverages the novel data format in a most trivial way. The code is scalar—that is, no explicit vectorization is performed.

[8]More details can be found at http://www.cscs.ch/computers/piz_daint/.
[9]For more information, see http://www.cscs.ch/computers/grand_tave/.

```
vector<double> xpv(xps.size(), 1.0);
for (int i = 0, e = xpv.size(); i < e; i++)
{
    const Index<uint16_t>& index = xps[i];
    const uint32_t& j = index.index;
    double xp = LinearBasis(x[j], index.l, index.i);
    xpv[i] = fmax(0.0, xp);
}

for (int i = 0, ichain = 0; i < nno; i++, ichain += nfreqs)
{
    double temp = 1.0;
    for (int ifreq = 0; ifreq < nfreqs; ifreq++)
    {
        const auto& idx = chains[ichain + ifreq];
        if (!idx) break;

        temp *= xpv[idx];
        if (!temp) goto zero;
    }

    for (int dof = 0; dof < ndofs; dof++)
        value[dof] += temp * surplus(i, dof);

zero :

    continue;
}
```

```
for (int i = 0; i < nno; i++)
{
    double temp = 1.0;
    for (int j = 0; j < DIM; j++)
    {
        double xp = LinearBasis(
            x[j], index(i, j).l, index(i, j).i↩
                );
        if (xp <= 0.0) goto zero;

        temp *= xp;
    }

    for (int dof = 0; dof < ndofs; dof++)
        value[dof] += temp * surplus(i, dof);

zero :

    continue;
}
```

Fig. 5: Comparison of the interpolation kernels for an x86 code with (left) and without (right) sparse grid index compression (cf. (14)).

version	"7k" test [sec]	"300k" test [sec]
gold	0.000820	0.018884
x86	0.000197	0.004251
avx	0.000204	0.004221
avx2	0.000204	0.004234
avx512	0.000225	0.000907
cuda	0.000122	0.000275

TABLE II: Performance of the interpolation kernels on various target platforms (time measured in seconds). Note that the runtime reported for the *cuda* version accounts both for the execution time of the kernel as well as the data transfers into the final value.

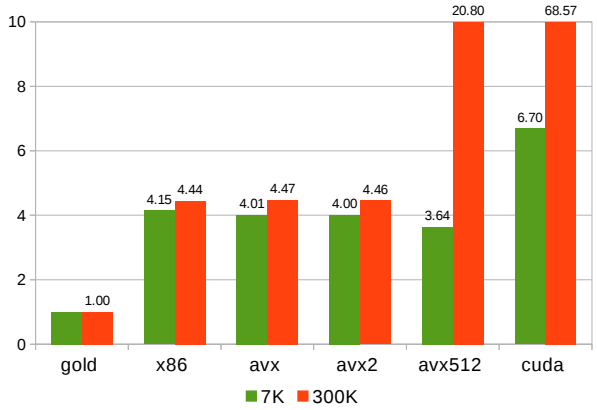


Fig. 6: Normalized speedup gains of various interpolation kernels for the two test cases (cf. Fig. I).

AVX/AVX2: In the *AVX/AVX2* kernels, the compute loops are manually vectorized. The *AVX2* additionally deploys vector FMA instructions where applicable. The effect of these optimizations is minimal due to the memory-bound nature of our problem.

AVX512: Unlike its *AVX/AVX2* siblings, the *AVX512* version has to deal with much less size of the cache per compute core. Therefore, it deploys OpenMP parallelization inside the interpolation kernel instead of high-level TBB work distribution (cf. Fig. 2). As long as the kernel performs the summing of the nno vectors, *AVX512* deploys an OpenMP 4 user-defined reduction with partial vector sums implemented in 512-bit wide intrinsics. By the nature of the algorithm, many partial vector sums end up making zero contributions. They are handled specially to initiate no actual memory flow and to reduce the cache pollution, yet causing imbalances in the reduction tree traversal.

CUDA: The *CUDA* version offloads the interpolation kernel to the NVIDIA Tesla P100 GPU. The scheduler uses a block size of 128, which is the closest to the $ndofs$ per point.[10] The nno is distributed across the maximum number of concurrent blocks for a given SM and register count. In this way, the whole kernel workload efficiently goes through in a single "wave" of blocks. The xpv array is mapped onto the shared memory. Unlike the "300k" test case, the "7k" benchmark is not sufficiently large to fully utilize the P100 compute resources and therefore demonstrates only a moderate speedup.

As an indicative performance measure, we consider the average execution time of a particular kernel. The data was generated by evaluating the interpolation kernels at $1,000$ randomly sampled grid points in $\mathbf{B}$ and then taking the average runtime. The performance results for the various implementations are reported in Tab. II and Fig. 6. The latter is normalized with respect to the *gold* version. Note that all kernels—except *AVX512* and *CUDA*—are single-threaded and

[10]Note that the variable $ndofs = 2 \cdot d = 118$ corresponds to the 118 coefficients $\alpha = (\alpha^k, \alpha^v)$ that are used to approximate the policy and value functions (see Sec. II-A).

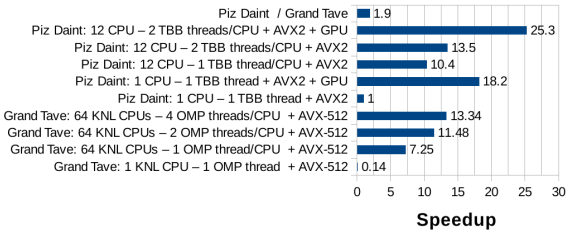


Fig. 7: Comparison of wall times for different stochastic OLG code variants on a single node of "Piz Daint" and "Grand Tave". The speedup is normalized with respect to an optimized, single-threaded test instance on "Piz Daint", whose runtime corresponds to 2,243 seconds.

therefore delegate the thread parallelism to the upper-level TBB scheduler (see Fig. 2). As shown in Fig. 6, we find that deploying the novel data structure delivers a speedup of about $4\times$, whereas in combination with $AVX512$ and $CUDA$ (where there are also more compute resources available), we can reach a combined speedup of almost two orders of magnitude. For further details, please refer to the interpolation kernels source code [25].

B. Single-node performance: KNL versus CPU/GPU clusters

To give a measure of how the single-node parallelization scheme discussed in Sec. IV-A impacts the performance, we evaluate the first two sparse grid levels of a single time step from the OLG model as outlined in Sec. II. This relatively small instance consists of $16 \cdot 119 = 1,904$ grid points, $16 \cdot 119 \cdot 59 = 112,336$ variables and constraints. The results are summarized in Fig. 7. They indicate a total speedup of $25\times$ when going from a single CPU thread implementation to a more efficient version of utilizing both all CPU and GPU resources present on a "Piz Daint" compute node. In case of running the same experiment on "Grand Tave", we find that utilizing Xeon Phi KNL in a multi-threaded mode delivers a speedup of about $96\times$ over a single-threaded version. Moreover, we observe that for our target application, "Piz Daint" nodes are about $2\times$ faster than the ones from "Grand Tave".

C. Strong Scaling

We now report the strong scaling efficiency of our code. The test problem is again a single time step of a 59-dimensional OLG model with 16 discrete, stochastic states. To provide a consistent benchmark, we used a nonadaptive sparse grid of refinement level 4 that was restarted from a sparse grid of level 2. This test case consists of $16 \cdot 281,077 = 4,497,232$ points and $16 \cdot 281,077 \cdot 59 = 265,336,688$ unknowns and constraints per time step. The economic test case was solved with increasingly larger numbers of nodes (from 1 to $4,096$ nodes on "Piz Daint"). Fig. 8 shows the to normalized execution time and scaling on different levels and their ideal speedups. We used 1 MPI process per multi-threaded node. In case of running the benchmark on "Piz Daint", the code scales nicely up to $4,096$ nodes, where the overall efficiency still is around

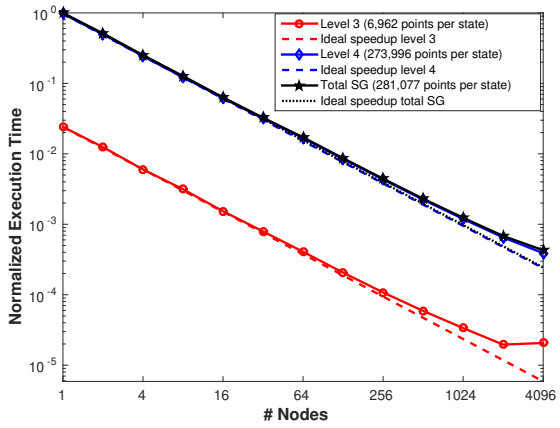


Fig. 8: Strong scaling on "Piz Daint" for an OLG model using 4 levels of grid refinements, 16 discrete states, and $16 \cdot 281,077 = 4,497,232$ points and $265,336,688$ unknowns in total. "Total SG" shows the entire, normalized simulation time up to $4,096$ nodes, where the runtime for a single node corresponds to $20,471$ seconds. We also show normalized execution times for the computational sub-components on different levels, e.g., for level 3 using 6.962 points. Dashed and dotted lines show the ideal speedup.

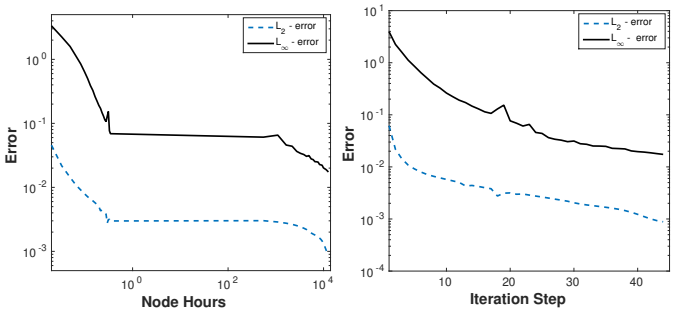


Fig. 9: Comparison of the L_2 and $L_\infty-$error for adaptive sparse grid solutions of the 59-dimensional OLG model as a function of compute time or iterations spent on "Piz Daint".

70%.[11] Thus, combined with the single-node speedup gains reported in section V-B we attain an overall speedup of more than four orders of magnitude for our benchmark. There is one dominant limitation to the strong scaling. It stems from the fact that within the lower refinement levels, the ratio of "points to be evaluated per thread" is often smaller than one with increasing node numbers, i.e., threads are idling. The better parallel efficiency on the higher refinement levels is due to the fact we have in this situation many more points available, resulting in a workload that is somewhat fairer distributed among the different MPI processes and their respective threads (see Fig. 8).

D. Convergence of the time iteration algorithm

For the purpose of testing the convergence of our massively parallel time iteration algorithm, we compute equilibria for the model outlined in section II and a decreasing refinement

[11] Note that due to the limited size of "Grand Tave"—less than 200 nodes—we did not add the corresponding strong scaling figures here. From Fig. 8, it becomes evident that our code scales almost perfectly on such a small system.

threshold ϵ (see section III). In the left panel of Fig. 9, we compare the decaying L_2- and L_∞- error for a complete simulation of a 59-dimensional model as a function of compute time. The right panel of Fig. 9 shows the decreasing errors as a function of iteration steps. It is apparent from Fig. 9 that convergence of the time iteration algorithm is rather slow. This is to be expected, as time iteration has, at best, a linear convergence rate in iterations [26]. The time iteration was terminated once the average error dropped below the satisfactory level of 0.1 percent. For this iteration step, the ASGs consist in average of $73,874$ points per state, however varying between a minimum of $69,026$ points in state $z = 6$ and a maximum of $76,645$ points in state $z = 1$.[12]

VI. Conclusions

Solving mixed high-dimensional continuous/discrete dynamic stochastic economic models in competitive times, that is, in hours or days of human time at maximum imposes many challenges both from a modeling as well as from a computational perspective. We demonstrate in this paper that by combining ASGs (that ameliorate the curse of dimensionality imposed by the large heterogeneity of the economic model) with efficient data structures, a time iteration algorithm (that deals with the recursive nature of the problem formulation), and with hybrid HPC compute paradigms (which drastically reduces the time-to-solution process), we can handle the difficulties imparted by this particular model class up to a level of complexity not seen before. By exploiting the generic structure of the economic model under consideration, we implemented a hybrid parallelization scheme that uses state-of-the-art parallel computing paradigms. It minimizes MPI interprocess communication by using TBB and AVX, AVX2 or AVX-512 vectorization (depending on the hardware available), and partially offloads the function evaluations to GPUs if available. In addition, we introduced a novel data compression scheme for ASGs that resulted in accelerating the compute time spent on interpolations by about $4\times$. Numerical tests on "Piz Daint" (a hybrid CPU/GPU system) and "Grand Tave" (a Xeon Phi KNL cluster) at CSCS show that our code is highly scalable. In the case of a stochastic public finance OLG model with 60 generations and sixteen discrete states, we found excellent strong scaling properties up to $4,096$ nodes, resulting in an overall speedup of more than four orders of magnitude compared to a single, optimized CPU thread.

Acknowledgements

All authors gratefully acknowledge financial support from PASC. This work was supported by a grant from the Swiss National Supercomputing Centre under project IDs s555 and s790.

[12]Note that we carried out our computations by setting $L_{max} = 6$ and fixing ϵ until the error level did not improve any further. We subsequently restarted the code with a decreased value of ϵ. This measure then slightly adds points to the grid and therefore further lowers the error. As the size of the classical sparse grid grows very fast in high dimensions when the level increases—from 119 (L=2), $7,081$ (L=3), $281,077$ (L=4), $8,378,001$ (L=5), to $> 2 \cdot 10^8$ (L=6)— adaptive sparse grids allow us to look at intermediate numbers of grid points.

References

[1] P. A. Diamond, "National debt in a neoclassical growth model," *American Economic Review*, vol. 55, no. 5, pp. 1126–1150, December 1965.
[2] R. W. Evans, L. J. Kotlikoff, and K. L. Phillips, "Game over: Simulating unsustainable fiscal policy," National Bureau of Economic Research, Working Paper 17917, March 2012.
[3] A. Auerbach and L. Kotlikoff, *Dynamic Fiscal Policy*. Cambridge University Press, 1987.
[4] M. Feldstein and J. Liebman, "Social security," National Bureau of Economic Research, Inc, NBER Working Papers 8451, 2001.
[5] D. Krueger and F. Kubler, "Pareto-Improving Social Security Reform when Financial Markets are Incomplete!?" *American Economic Review*, vol. 96, no. 3, pp. 737–755, 2006.
[6] K. L. J. David S. Bizer, "Taxation and uncertainty," *The American Economic Review*, vol. 79, no. 2, pp. 331–336, 1989.
[7] D. Krueger and F. Kubler, "Computing equilibrium in OLG models with stochastic production," *Journal of Economic Dynamics and Control*, vol. 28, no. 7, pp. 1411–1436, 2004.
[8] J. Hasanhodzic and L. J. Kotlikoff, "Generational risk - is it a big deal?: Simulating an 80-period olg model with aggregate shocks," National Bureau of Economic Research, Working Paper 19179, June 2013.
[9] K. L. Judd, *Numerical methods in economics*. The MIT press, 1998.
[10] H.-J. Bungartz and M. Griebel, "Sparse grids," *Acta Numerica*, vol. 13, pp. 1–123, 2004.
[11] X. Ma and N. Zabaras, "An adaptive hierarchical sparse grid collocation algorithm for the solution of stochastic differential equations," *J. Comput. Phys.*, vol. 228, no. 8, pp. 3084–3113, 2009.
[12] D. Pflüger, "Spatially adaptive refinement," in *Sparse Grids and Applications*, ser. Lecture Notes in Computational Science and Engineering, J. Garcke and M. Griebel, Eds. Berlin Heidelberg: Springer, 2012, pp. 243–262.
[13] F. Nobile, R. Tempone, and C. G. Webster, "A sparse grid stochastic collocation method for partial differential equations with random input data," *SIAM Journal on Numerical Analysis*, vol. 46, no. 5, pp. 2309–2345, 2008.
[14] A. Skjellum, W. Gropp, and E. Lusk, *Using MPI*. MIT Press, 1999.
[15] J. Reinders, *Intel Threading Building Blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.
[16] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: Stream computing on graphics hardware," in *ACM SIGGRAPH 2004 Papers*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004, pp. 777–786.
[17] J. Brumm and S. Scheidegger, "Using adaptive sparse grids to solve high-dimensional dynamic models," *Econometrica*, vol. 85, no. 5, pp. 1575–1612, 2017.
[18] J. Brumm, D. Mikushin, S. Scheidegger, and O. Schenk, "Scalable high-dimensional dynamic stochastic economic modeling," *Journal of Computational Science*, vol. 11, pp. 12 – 25, 2015.
[19] N. Stokey, R. Lucas, and E. Prescott, "Recursive methods in economic dynamics," *Cambridge MA*, 1989.
[20] J. Garcke and M. Griebel, *Sparse Grids and Applications*, ser. Lecture Notes in Computational Science and Engineering Series. Springer, 2012.
[21] R. Bellman, *Adaptive Control Processes: A Guided Tour*, ser. Rand Corporation. Research studies. Princeton University Press, 1961.
[22] H.-J. Bungartz and S. Dirnstorfer, "Multivariate quadrature on adaptive sparse grids," *Computing*, vol. 71, pp. 89–114, 2003.
[23] A. Heinecke and D. Pflueger, "Emerging architectures enable to boost massively parallel data mining using adaptive sparse grids," *International Journal of Parallel Programming*, vol. 41, no. 3, pp. 357–399, 2013.
[24] A. Waechter and L. T. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Math. Program.*, vol. 106, no. 1, pp. 25–57, 2006.
[25] S. Scheidegger and D. Mikushin, "Interpolation backends for hddm-solver," https://github.com/apc-llc/hddm-solver-postprocessors/tree/2016v1_olgtax, 2018, [Online; accessed 7-January-2018].
[26] W. L. Maldonado and B. Svaiter, "Hölder continuity of the policy function approximation in the value function approximation," *Journal of Mathematical Economics*, vol. 43, no. 5, pp. 629–639, 2007.